HW5

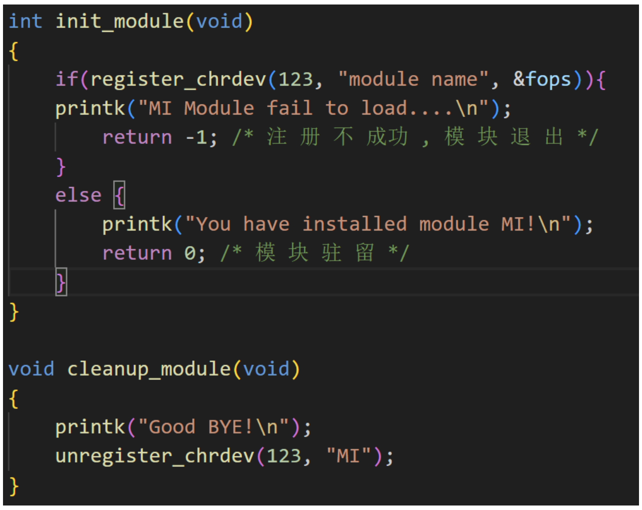
黄佳溢

December 17, 2024

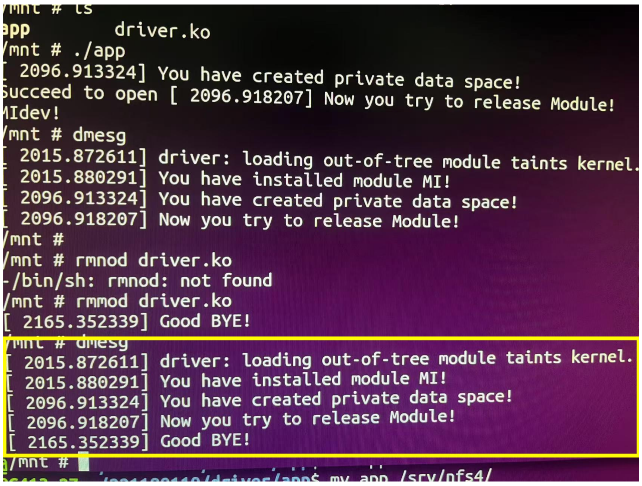
1 驱动程序逻辑

1.1 初始化Init和删除Cleanup操作

Init程序会在insmod时进行加载，主要用于**申请设备号和内核注册**，调用register_chrdev即可。而cleanup程序在rmmod时进行加载，调用unregister_chrdev即可。总体上没有什么难度，记得使用printk函数辅助debug。



```
int init_module(void)
{
    if(register_chrdev(123, "module name", &fops)){
    printk("MI Module fail to load....\n");
        return -1; /* 注 册 不 成功 , 模 块 退 出 */
    }
    else {
        printk("You have installed module MI!\n");
        return 0; /* 模 块 驻 留 */
    }
}

void cleanup_module(void)
{
    printk("Good BYE!\n");
    unregister_chrdev(123, "MI");
}
```



```
/mnt # ls
app          driver.ko
/mnt # ./app
[ 2096.913324] You have created private data space!
Succeed to open [ 2096.918207] Now you try to release Module!
MIdev!
/mnt # dmesg
[ 2015.872611] driver: loading out-of-tree module taints kernel.
[ 2015.880291] You have installed module MI!
[ 2096.913324] You have created private data space!
[ 2096.918207] Now you try to release Module!
/mnt #
/mnt # rmnod driver.ko
-/bin/sh: rmnod: not found
/mnt # rmmod driver.ko
[ 2165.352339] Good BYE!
/mnt # dmesg
[ 2015.872611] driver: loading out-of-tree module taints kernel.
[ 2015.880291] You have installed module MI!
[ 2096.913324] You have created private data space!
[ 2096.918207] Now you try to release Module!
[ 2165.352339] Good BYE!
/mnt # 
```

Figure 1: 具体程序

Figure 2: 结果

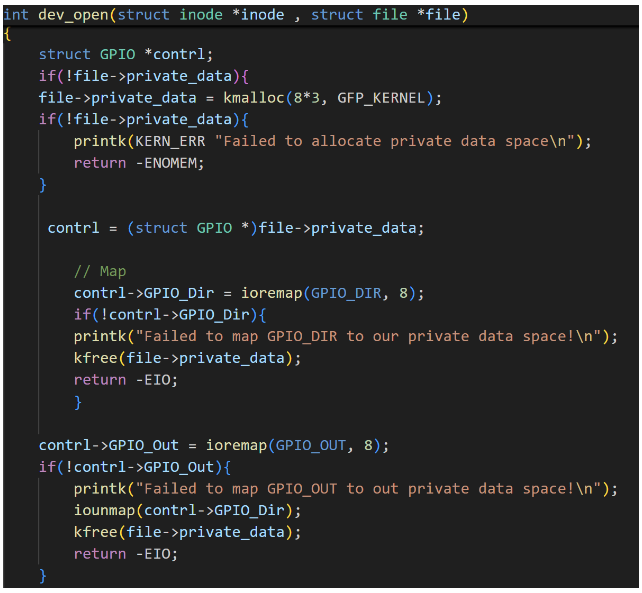
1.2 打开Open和关闭Release操作

使用insmod，我们的驱动程序完成初始化和内核注册之后，内核将会记录该驱动程序所申请到的设备号和操作函数执政fops。但是想要被应用程序所引用，需要使

用mknod，在VFS中为其生成设备文件/dev/MIdev，这样一来，我们的应用程序只需要使用open("/dev/MIdev", O_RDONLY)便可以调用操作函数。

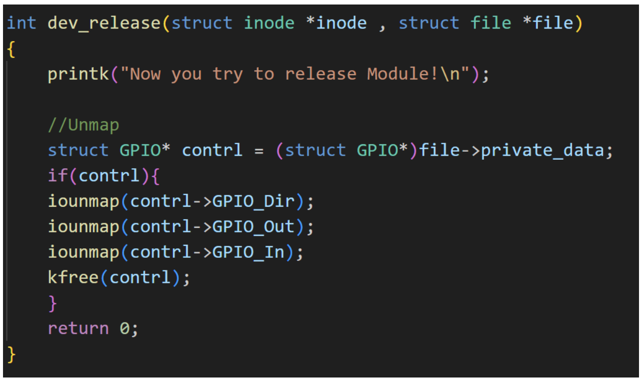
对于驱动程序的开发，应用程序每执行一次open函数，就会调用一次驱动open函数并根据open权限来建立专属的file结构体。**如果要实现后续的GPIO控制功能，就必须申请file→private_data，并在里面完成GPIO控制寄存器的地址映射。**

由于每一个寄存器都是64位（8个字节），共3个（GPIO_DIR, GPIO_OUT, GPIO_IN），所以我们需要申请$8 \times 3 = 24$位内核空间。然后嵌套上struct结构并进行寄存器地址映射。



```
int dev_open(struct inode *inode , struct file *file)
{
    struct GPIO *contrl;
    if(!file->private_data){
    file->private_data = kmalloc(8*3, GFP_KERNEL);
    if(!file->private_data){
        printk(KERN_ERR "Failed to allocate private data space\n");
        return -ENOMEM;
    }

     contrl = (struct GPIO *)file->private_data;

        // Map
        contrl->GPIO_Dir = ioremap(GPIO_DIR, 8);
        if(!contrl->GPIO_Dir){
        printk("Failed to map GPIO_DIR to our private data space!\n");
        kfree(file->private_data);
        return -EIO;
        }

    contrl->GPIO_Out = ioremap(GPIO_OUT, 8);
    if(!contrl->GPIO_Out){
        printk("Failed to map GPIO_OUT to out private data space!\n");
        iounmap(contrl->GPIO_Dir);
        kfree(file->private_data);
        return -EIO;
    }
```

Figure 3: open函数（一部分）



```
int dev_release(struct inode *inode , struct file *file)
{
    printk("Now you try to release Module!\n");

    //Unmap
    struct GPIO* contrl = (struct GPIO*)file->private_data;
    if(contrl){
    iounmap(contrl->GPIO_Dir);
    iounmap(contrl->GPIO_Out);
    iounmap(contrl->GPIO_In);
    kfree(contrl);
    }
    return 0;
}
```

Figure 4: release函数

应用程序每执行一个close函数，就会调用一个驱动release函数并删除对应的file结构体，在release函数中，我们需要**①取消设备号 ②取消地址映射 ③归还内存**。

1.3 读Read写Write操作以及IO控制ioctl操作

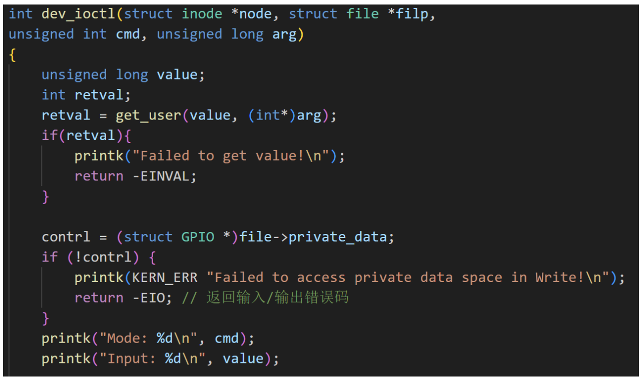
由于我们需要实现GPIO端口的控制功能，所以**这里仅展示ioctl函数，后面还有write函数的函数实现。**

1.3.1 ioctl函数具体流程

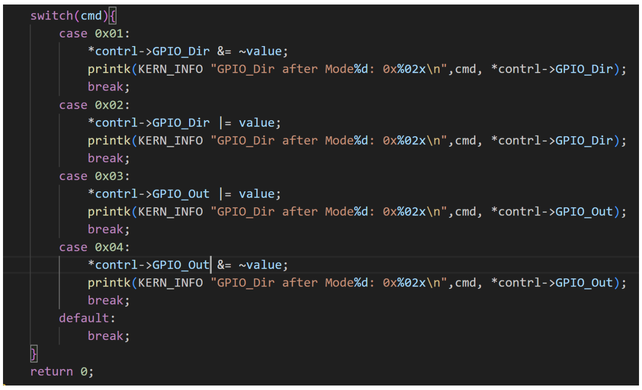
1. $get_user(value, (int*arg))$获得应用程序调用时输入的参数$value$，$value$是$unsigned\ long$八位数据，**用于寄存器赋值。**

2. 判断cmd，将ioctl函数分成四种模式：①$cmd = 0x01$，此时将$GPIO_Dir \&= \ value$；②$cmd = 0x02$，此时将$GPIO_Dir \,| = value$；③$cmd = 0x03$，此时将$GPIO_Out \,| = value$；④$cmd = 0x04$，此时将$GPIO_Out \ \&= \ value$

总体上的实现还是非常简易易懂的，唯一需要注意的点就是get_user别忘了。



```c
int dev_ioctl(struct inode *node, struct file *filp,
unsigned int cmd, unsigned long arg)
{
    unsigned long value;
    int retval;
    retval = get_user(value, (int*)arg);
    if(retval){
        printk("Failed to get value!\n");
        return -EINVAL;
    }

    contrl = (struct GPIO *)file->private_data;
    if (!contrl) {
        printk(KERN_ERR "Failed to access private data space in Write!\n");
        return -EIO; // 返回输入/输出错误码
    }
    printk("Mode: %d\n", cmd);
    printk("Input: %d\n", value);
```



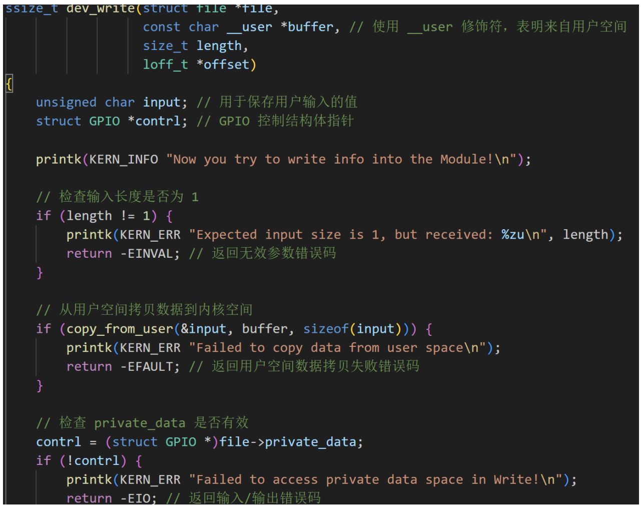
```c
    switch(cmd){
        case 0x01:
            *contrl->GPIO_Dir &= ~value;
            printk(KERN_INFO "GPIO_Dir after Mode%d: 0x%02x\n",cmd, *contrl->GPIO_Dir);
            break;
        case 0x02:
            *contrl->GPIO_Dir |= value;
            printk(KERN_INFO "GPIO_Dir after Mode%d: 0x%02x\n",cmd, *contrl->GPIO_Dir);
            break;
        case 0x03:
            *contrl->GPIO_Out |= value;
            printk(KERN_INFO "GPIO_Dir after Mode%d: 0x%02x\n",cmd, *contrl->GPIO_Out);
            break;
        case 0x04:
            *contrl->GPIO_Out &= ~value;
            printk(KERN_INFO "GPIO_Dir after Mode%d: 0x%02x\n",cmd, *contrl->GPIO_Out);
            break;
        default:
            break;
    }
    return 0;
```

Figure 5: ioctl函数截图一

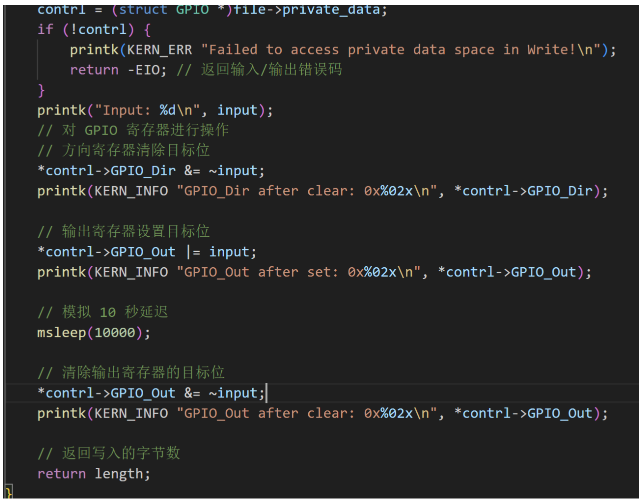
Figure 6: ioctl函数截图二

1.3.2 write函数具体流程

这部分思想和ioctl函数相似，所以也便不详细解释，这里只展示源码。



```c
ssize_t dev_write(struct file *file,
                  const char __user *buffer, // 使用 __user 修饰符，表明来自用户空间
                  size_t length,
                  loff_t *offset)
{
    unsigned char input; // 用于保存用户输入的值
    struct GPIO *contrl; // GPIO 控制结构体指针

    printk(KERN_INFO "Now you try to write info into the Module!\n");

    // 检查输入长度是否为 1
    if (length != 1) {
        printk(KERN_ERR "Expected input size is 1, but received: %zu\n", length);
        return -EINVAL; // 返回无效参数错误码
    }

    // 从用户空间拷贝数据到内核空间
    if (copy_from_user(&input, buffer, sizeof(input))) {
        printk(KERN_ERR "Failed to copy data from user space\n");
        return -EFAULT; // 返回用户空间数据拷贝失败错误码
    }

    // 检查 private_data 是否有效
    contrl = (struct GPIO *)file->private_data;
    if (!contrl) {
        printk(KERN_ERR "Failed to access private data space in Write!\n");
        return -EIO; // 返回输入/输出错误码
```



```c
    contrl = (struct GPIO *)file->private_data;
    if (!contrl) {
        printk(KERN_ERR "Failed to access private data space in Write!\n");
        return -EIO; // 返回输入/输出错误码
    }
    printk("Input: %d\n", input);
    // 对 GPIO 寄存器进行操作
    // 方向寄存器清除目标位
    *contrl->GPIO_Dir &= ~input;
    printk(KERN_INFO "GPIO_Dir after clear: 0x%02x\n", *contrl->GPIO_Dir);

    // 输出寄存器设置目标位
    *contrl->GPIO_Out |= input;
    printk(KERN_INFO "GPIO_Out after set: 0x%02x\n", *contrl->GPIO_Out);

    // 模拟 10 秒延迟
    msleep(10000);

    // 清除输出寄存器的目标位
    *contrl->GPIO_Out &= ~input;
    printk(KERN_INFO "GPIO_Out after clear: 0x%02x\n", *contrl->GPIO_Out);

    // 返回写入的字节数
    return length;
}
```

Figure 7: write函数截图一

Figure 8: write函数截图二

2 实验结果

我选用$GPIO2$作为实验端口，查询实验手册得到插板的输出引脚，通过示波器看到观测实验结果：

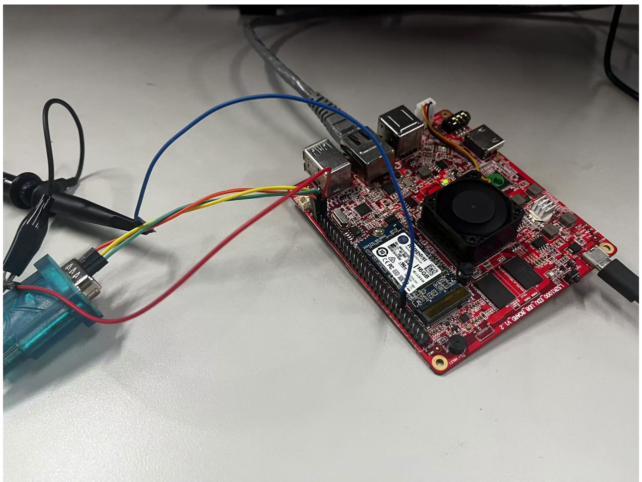
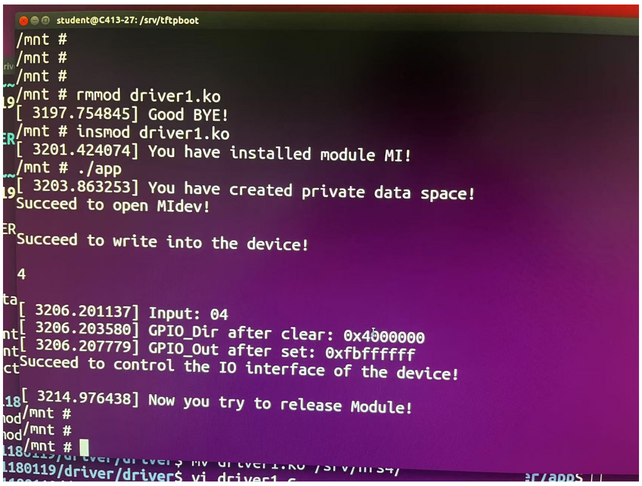


Figure 9: 电路连线



```
student@C413-27: /srv/tftpboot
/mnt #
/mnt #
/mnt #
/mnt # rmmod driver1.ko
[ 3197.754845] Good BYE!
/mnt # insmod driver1.ko
[ 3201.424074] You have installed module MI!
/mnt # ./app
[ 3203.863253] You have created private data space!
Succeed to open MIdev!

Succeed to write into the device!

4

[ 3206.201137] Input: 04
[ 3206.203580] GPIO_Dir after clear: 0x4000000
[ 3206.207779] GPIO_Out after set: 0xfbffffff
Succeed to control the IO interface of the device!

[ 3214.976438] Now you try to release Module!
/mnt #
/mnt #
/mnt #
```

Figure 10: 程序输出

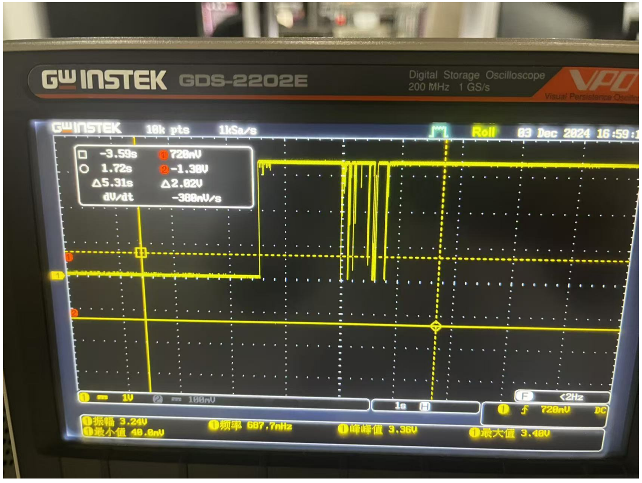


Figure 11: 观测到高电平降到低电平

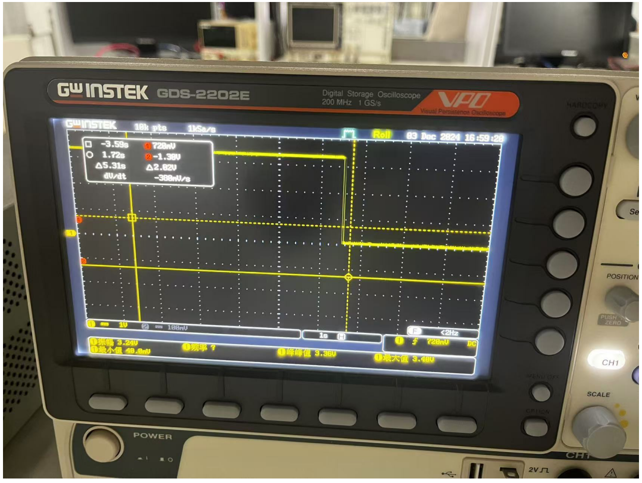
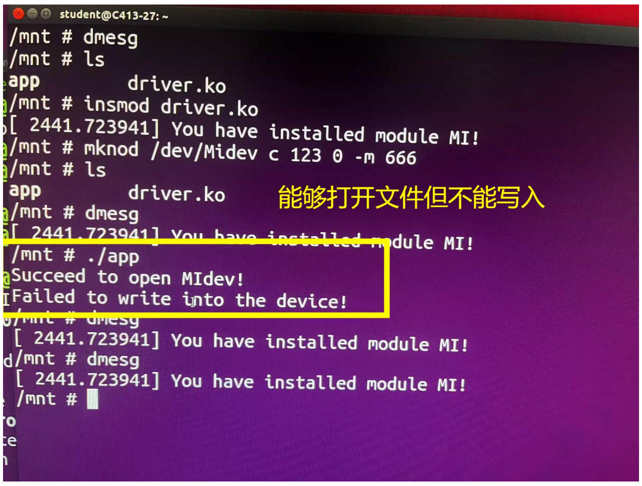


Figure 12: 观测到低电平升到高电平

3 踩坑

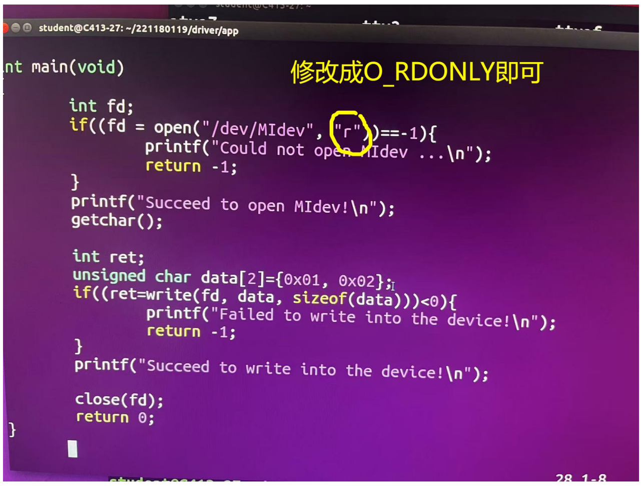
1. **打开设备文件时，访问权限不是r和w，而是O_RDONLY和O_WRONLY**
   在测试的过程中，我们都会涉及到对设备的打开和读写，此时需要保证自己的权限足够。



```
student@C413-27: ~
/mnt # dmesg
/mnt # ls
app          driver.ko
/mnt # insmod driver.ko
[ 2441.723941] You have installed module MI!
/mnt # mknod /dev/Midev c 123 0 -m 666
/mnt # ls
app          driver.ko
/mnt # dmesg
[ 2441.723941] You have installed module MI!
/mnt # ./app
Succeed to open MIdev!
Failed to write into the device!
/mnt # dmesg
[ 2441.723941] You have installed module MI!
/mnt # dmesg
[ 2441.723941] You have installed module MI!
/mnt # 
```

能够打开文件但不能写入

Figure 13: 报错情况



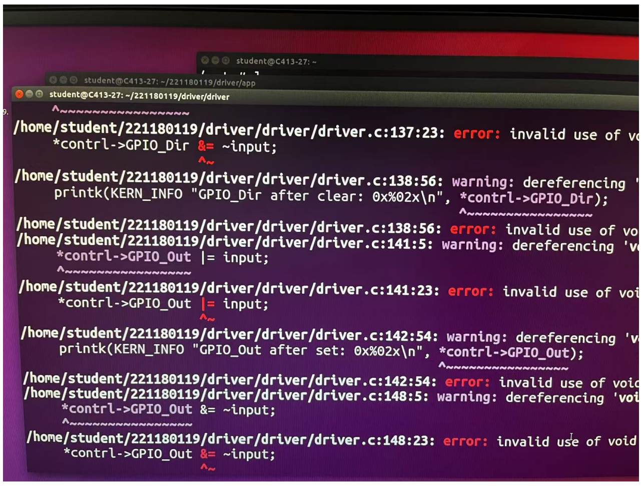
```
student@C413-27: ~/221180119/driver/app
nt main(void)

        int fd;
        if((fd = open("/dev/MIdev", "r"))==-1){
                printf("Could not open MIdev ...\n");
                return -1;
        }
        printf("Succeed to open MIdev!\n");
        getchar();

        int ret;
        unsigned char data[2]={0x01, 0x02};
        if((ret=write(fd, data, sizeof(data)))<0){
                printf("Failed to write into the device!\n");
                return -1;
        }
        printf("Succeed to write into the device!\n");

        close(fd);
        return 0;
}
                                                            28,1-8
```

修改成O_RDONLY即可

Figure 14: 解决方法

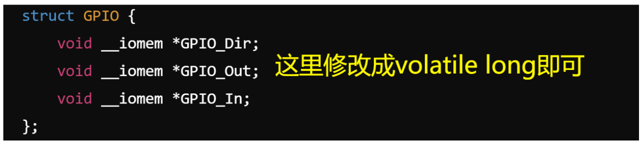
2. **在执行位与或操作时，操作数类型不能为void**

这个错误其实是我自找的QAQ，一开始使用的代码是由GPT自动生成的，所以contrl→GPIO_Dir的类型被设置为void __iomem *，此时只需要修改成volatile long即可。



```
student@C413-27: ~/221180119/driver/driver
    ^~~~~~~~~~~~~~~~~~
/home/student/221180119/driver/driver/driver.c:137:23: error: invalid use of vo
    *contrl->GPIO_Dir &= ~input;
                      ^~
/home/student/221180119/driver/driver/driver.c:138:56: warning: dereferencing '
    printk(KERN_INFO "GPIO_Dir after clear: 0x%02x\n", *contrl->GPIO_Dir);
                                                       ^~~~~~~~~~~~~~~~~~
/home/student/221180119/driver/driver/driver.c:138:56: error: invalid use of vo
/home/student/221180119/driver/driver/driver.c:141:5: warning: dereferencing 'v
    *contrl->GPIO_Out |= input;
    ^~~~~~~~~~~~~~~~~~
/home/student/221180119/driver/driver/driver.c:141:23: error: invalid use of voi
    *contrl->GPIO_Out |= input;
                      ^~
/home/student/221180119/driver/driver/driver.c:142:54: warning: dereferencing 'v
    printk(KERN_INFO "GPIO_Out after set: 0x%02x\n", *contrl->GPIO_Out);
                                                     ^~~~~~~~~~~~~~~~~~
/home/student/221180119/driver/driver/driver.c:142:54: error: invalid use of voi
/home/student/221180119/driver/driver/driver.c:148:5: warning: dereferencing 'voi
    *contrl->GPIO_Out &= ~input;
    ^~~~~~~~~~~~~~~~~~
/home/student/221180119/driver/driver/driver.c:148:23: error: invalid use of void
    *contrl->GPIO_Out &= ~input;
                      ^~
```

Figure 15: 报错情况



```
struct GPIO {
    void __iomem *GPIO_Dir;
    void __iomem *GPIO_Out;
    void __iomem *GPIO_In;
};
```

这里修改成volatile long即可

Figure 16: 解决方法

4 代码

所有的代码已经上传至附件。